Midterm Exam

```
 _____ _____ ____ ____    _____ _____ ___  
| ____| ____/ ___/ ___|  |___ /|___  / _ \ 
|  _| |  _|| |   \___ \    |_ \   / / | | |
| |___| |__| |___ ___) |  ___) | / /| |_| |
|_____|_____\____|____/  |____/ /_/  \___/ 
```

EECS 370 Winter 2022: Intro to Computer Organization

You are to abide by the University of Michigan College of Engineering Honor Code. Please sign below to signify that you have kept the honor code pledge:

I have neither given nor received aid on this exam,
nor have I concealed any violations of the Honor Code.

Signature: ______________________________

Name: ______________________________

Uniqname: ______________________________

First/Last name of person sitting to your ***Right***
(Write ⊥ if you are at the end of the row) ______________________________

First/Last name of person sitting to your ***Left***
(Write ⊥ if you are at the end of the row) ______________________________

Exam Directions:

- You have **120 minutes** to complete the exam. There are **9** questions in the exam on **16** pages. **Please flip through your exam and ensure that you have all 16 pages.**
- You must show your work to be eligible for partial credit!
- Write legibly and dark enough for the scanners to read your answers.
- **Write your uniqname on the line provided at the top of each page.**

Exam Materials:

- You are allotted **one 8.5 x 11 double-sided** note sheet to bring into the exam room.
- The **ARM Reference Sheet** has been handed out as supplemental exam material**.**
- You are allowed to use calculators that do not have an internet connection. All other electronic devices, such as cell phones or anything or calculators with an internet connection, are strictly forbidden and usage will result in an Honor Code violation.

1.	Short Questions	________	/	15 pts
2.	Caller/Callee Save	________	/	10 pts
3.	Linker	________	/	10 pts
4.	Understanding the LC2K ISA	________	/	10 pts
5.	LC2K Processor Performance	________	/	9 pts
6.	LC2K Multi-Cycle Datapath	________	/	12 pts
7.	Debugging Multi-Cycle Datapath	________	/	10 pts
8.	ISA Design	________	/	12 pts
9.	Convert C to ARM Assembly	________	/	12 pts
	TOTAL	________	/	100 pts

1.	**Short Questions** **[15 pts]**
	Complete the following true/false and short answer questions.

True/False Questions — Circle One:

(a) **[1 pt]** A little endian system stores the most significant byte of a data word at the highest memory address. True / False

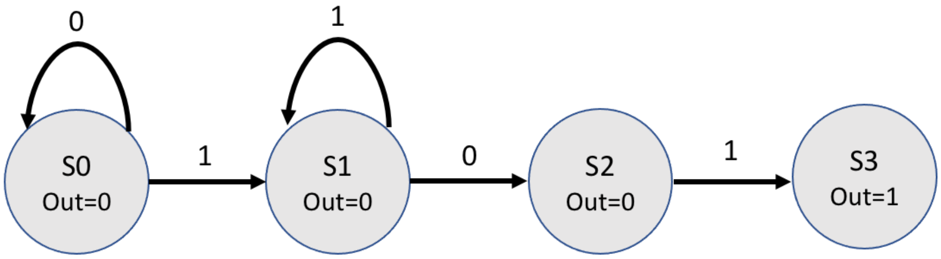
(b) **[1 pt]** All numeric values represented by a 32-bit two's complement integer can be represented by a 32-bit IEEE 754 single precision number. True / False

(c) **[1 pt]** Static variables are allocated to stack. True / False

(d) **[1 pt]** For leaf function it is always more efficient to allocate caller save registers if they are available. Recall that leaf functions are those that don't call any other functions. True / False

Short Answer Questions

(e) **[4 pts]** A programmer wanted to design a fast FSM to detect a 3-digit binary sequence "101". See the incomplete FSM below. It misses **3 arrows** or state transitions. Please complete the state machine.



Is this a Mealy or Moore machine? ____________________

(f) **[2 pts]** How many bits of ROM memory are needed to design a controller for a finite state machine with 32 states, 3 input bits, and 11 output bits? Please show your work!

(g) **[3 pts]** Design a 2-input OR gate using only 2-input NAND gates. Draw the logic here:

Assuming the delay of the NAND gate is 5 ns, what is the delay of the OR digital logic circuit above?

__________________ ns

(h) **[2 pts]** Compute the size of the **`simple_struct`**. Its size can be reduced by rearranging the position of **`char a[3]`** in the struct declaration. Calculate the optimal reduced size.

```
struct {
     char a[3];
     double b;
     int c;
} simple_struct;
```

Original size: ______________ bytes

Reduced size: ______________ bytes

2. Caller/Callee Save [10 pts]

Complete the caller/callee saved registers for the following C program.

Count the total number of executed caller/callee saves for the code shown when **`InExam()`** is called once. Assume the compiler checks for liveness across function calls **but does not make any other optimizations.**

```
void InExam()
{ ← a:1
    int a = 3, b = 7, c = 0;
    for (int i=0; i < a; i++) {
        foo();
    }
    b = a + 2;
    foo();
    c++;
    foo();
    a = b + a;
    foo();
} ↖
```

```
 void foo()
{
     int d = 2, e = 8, f = 1;
     printf("%d", d);
     e--;
     while (f < 5) {
         printf("%d", f);
          f++;
     }
     e = f * 2;
     printf("%d", e);
     e++;
 }
```

Determine the number of load/store instructions required for each variable in each case in the table below. In the fourth column of the table, assign variables to a caller or callee register to minimize the number of loads/stores executed. Assume there are only **2 callee** and **2 caller** registers.

Variable	Caller Save	Callee Save	Caller or Callee?
a	5	1	Callee.
b	2	1	Caller.
c	4	1	Callee
i	3	1	Caller
d	0	6	Caller.
e	2*6:12	6	Callee
f	5*6=30.	6	Callee

3.	**Linker** **[10 pts]**
	Complete the symbol and relocation tables for the following C program.

(a) **[8 pts]** Fill in the rest of the symbol and relocation tables for the following two C files. Note that not all entries in the tables below may be used. The tables are located on the next page.

burrito.c

```
1   extern void Helper(char*, int);
2
3   #define MAX_TOPPINGS 100
4
5   extern int location;
6   char Burrito[MAX_TOPPINGS][10];
7
8
9
10  void MakeBurrito(int   num_toppings,
11                   char* toppings[],
12                   int   width)
13  {
14    if(num_toppings > MAX_TOPPINGS){
15      // That’s one wild burrito!
16      return;
17    }
18
19
20    for(int i=0; i < num_toppings; i++){
21      Helper(toppings[i], width);
22      location += 1;
23    }
24  }
25
```

helper.c

```
1   #include "string.h"  // has strcpy(.)
2   #include “grocery.h” // has Buy(.)
3
4
5   extern char* Burrito[][10];
6   int location = 0;
7
8
9
10  void Helper(char* topping, int width)
11  {
12    Buy(topping, 5 * width);
13    strcpy(Burrito[location], topping);
14  }
15
16
17
18
19
20
21
22
23
24
25
```

burrito.o Symbol Table	
Symbol	**Type (T/D/U)**

helper.o Symbol Table	
Symbol	**Type (T/D/U)**

burrito.o Relocation Table		
Line	**Instruction (LDUR, STUR, BL)**	**Symbol**

helper.o Relocation Table		
Line	**Instruction (LDUR, STUR, BL)**	**Symbol**

(b) **[2 pt]** You have burrito.o and helper.o from a previous compilation. You change the initialization of **`int location = 0`** on **Line 5** of helper.c to a different integer value and create an executable containing this change. Which files, if any, must be re-compiled? Which files, if any, must be re-linked?

4. Understanding the LC2K ISA [10 pts]

Simulate an LC2K assembly program and answer questions about its operation.

You've discovered this mysterious LC2K assembly file and are tasked to understand what it computes. Simulate the program to understand its functionality. Assume that the return value is stored in **r5**. You can use this page to trace the program's execution, then answer the questions on the next page.

```
          lw      0       1       ArrPtr
          lw      0       2       ArrLen
          lw      0       3       Num
          add     0       0       5
loop      add     1       5       6           //r6 is array address
          lw      6       6       0           //ld array element
          beq     6       3       end
          lw      0       6       One
          add     5       6       5           //increment r5
          beq     2       5       done        //check loop condition
          beq     0       0       loop        //continue loop
done      lw      0       5       NegOne
end       halt
Num       .fill   6
One       .fill   1
NegOne    .fill   -1
ArrLen    .fill   4
ArrPtr    .fill   Array
Array     .fill   0
          .fill   3
          .fill   6
          .fill   2
```

(a) **[2 pts]** How many times will the label "`loop`" be executed during this program?

(b) **[2 pts]** What is the value of register `r5` and `r6` when the program halts?

(c) **[2 pts]** In one sentence, describe what this program is doing? (*Hint*: What does the return value indicate?)

(d) **[2 pts]** If `Num` had the value 8, what would this program return? What does this return value indicate?

(e) **[2 pts]** This program is somewhat inefficient. How could this program, specifically the body of the loop, be improved? Briefly describe a change you would make to the program that would make the loop more efficient. Your optimization should be *independent* of the content and size of `Array`.

5. LC2K Processor Performance [9 pts]

Compare performance of different LC2K processor datapaths studied in class

Consider the LC2K single-cycle, multi-cycle, and 5-stage pipelined datapaths from lecture. The datapath has the following delays:

`Read register file:`	**`10 ns`**
`Write register file:`	**`10 ns`**
`ALU:`	**`10 ns`**
`Read memory:`	**`50 ns`**
`Write memory:`	**`10 ns`**
`All other operations:`	**`0 ns`**

(a) **[3 pts]** Calculate the minimum **clock period in ns** for following designs:

Single-Cycle	**Multi-Cycle**	**5-Stage Pipeline**

(b) **[3 pts]** An engineer tries to run a program with only these instructions: 50 adds, 50 stores, and an unknown number of loads. The program takes 750 cycles for a multi-cycle processor. How many load instructions are in this program?

(c) **[3 pts]** What is the execution time in ns for this program on a 5-stage LC2K pipeline processor with the clock period calculated in part (a)? Assume there are no hazards.

6. LC2K Multi-Cycle Datapath [12 pts]

Extend the LC2K Multi-Cycle Datapath to compute new instructions!

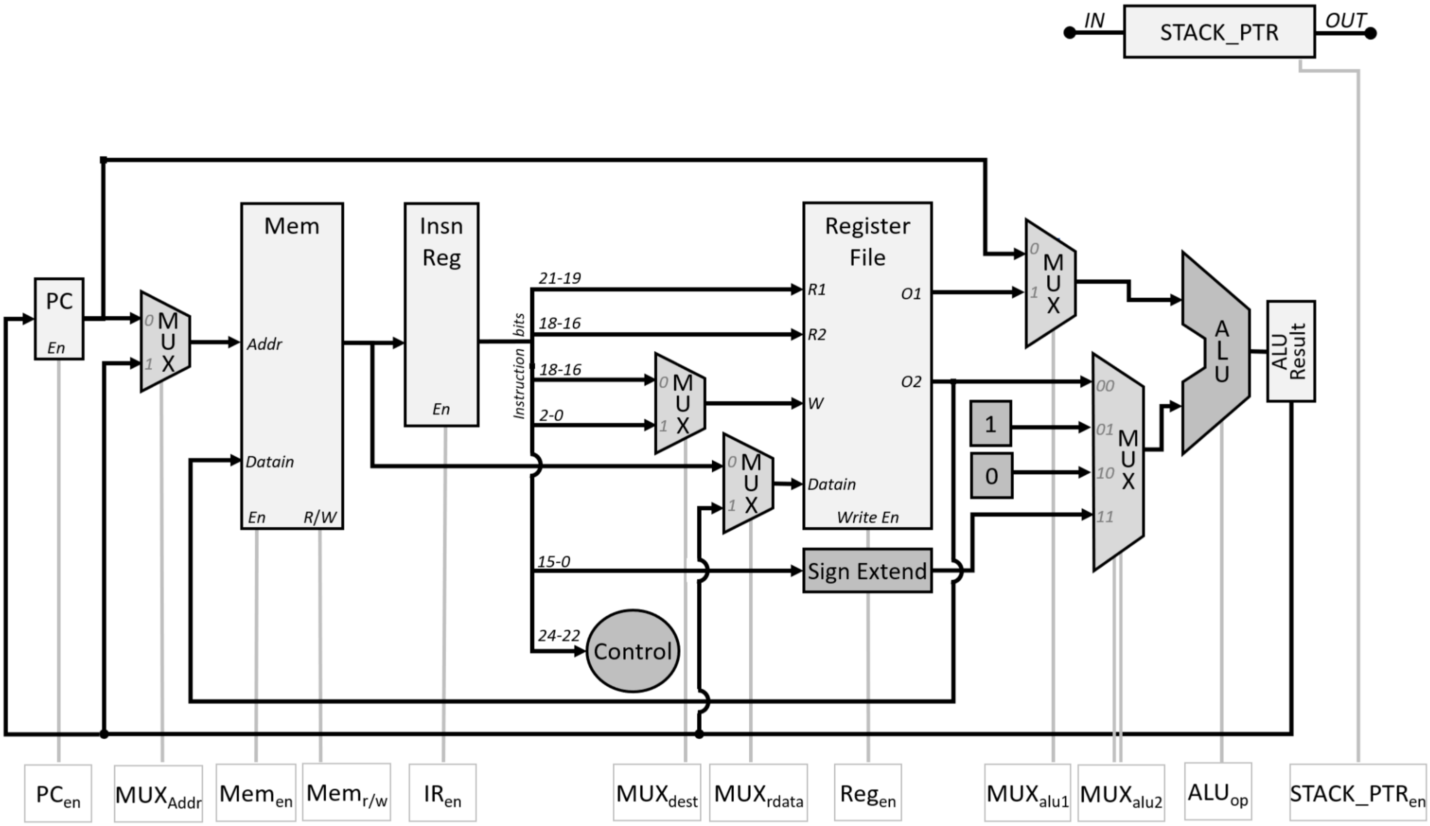
We want to extend the original LC2K multi-cycle datapath by supporting stack functionality. In order to achieve this, a new register called **STACK_PTR** has been added to the datapath. Our goal in this question is to modify the original datapath to support both new stack instructions as well as original LC2K instructions.

The new instructions for the stack include:

- ***SETSP regA*** STACK_PTR = regA
- ***PUSH regB*** (MEM [STACK_PTR] = regB; STACK_PTR = STACK_PTR + 1)
- ***POP regB*** (STACK_PTR = STACK_PTR - 1; regB = MEM [STACK_PTR])

Here are the specifications for the new instructions that we should support:

- Stack space lives inside the memory along with the machine codes and program data.
- Stack **grows towards higher** memory addresses.
- **STACK_PTR** would always point to the **next available** spot in the stack.
- **STACK_PTR** register would be updated when the **STACK_PTR**$_{\textbf{En}}$ signal is 1 (similar to register file).



(a) **[6 pts]** On the previous page, complete the design of the new datapath to support the stack functions. Draw the necessary connections. Your design should enable the **least number of cycles** for each new instruction. Note that we can extend any of the existing MUXes, but we can **only add a single new MUX. Only one of the MUXes can have more than 4 inputs**. New constants can also be added to the MUXes.

(b) **[6 pts]** Specify the operations for the remaining cycles of the **POP** instruction in the table below. Provide the exact changes to the multi-cycle datapath state. Assume **`[DATA_REG]`** stores the value read from memory, and **`[STACK_PTR]`** stores the value of the stack pointer register. You do not need to fill in all the blanks.

POP
Cycle 1

`[Instruction_Reg] = Mem[PC]`
`[ALU_Result] = PC + 1`

Cycle 2

`[PC] = [ALU_Result]`
`Read Register Values`

Cycle 3
______________________ = ______________________ ______________________ = ______________________
Cycle 4
______________________ = ______________________ ______________________ = ______________________
Cycle 5
______________________ = ______________________ ______________________ = ______________________

7. Debugging Multi-Cycle Datapath [10 pts]

Find fabrication bugs in the LC2K Multi-Cycle Datapath!

(a) **[6 pts]** During the fabrication of the LC2K multicycle datapath, we notice that there is a **bug in the control signals**. There is a bridge connection between ***PC_{en}*** and ***MUX_{rdata}*** signals. As a result of this, whenever one of the signals is set to have a high value, the other signal also gets a high value.

To expose the bug in the datapath, the following test case has been written and the datapath is verified with different values of **X** and **Y**. Please circle the lines of code that could potentially expose the bug in the datapath, and briefly explain how those lines could expose the bug.

```
        lw     0   1   num1
        lw     0   2   num2
        add    1   2   3
        add    1   3   4
        beq    0   4   end
        sw     0   4   result
end     halt
num1    .fill  X
num2    .fill  Y
result  .fill  0
```

Explanation:

(b) **[4 pts]** For what value of **X** and **Y** would this test case ***not*** expose the bridge bug (*i.e.,* When the test case would behave the same as being run on the correct datapath)? Please show your work.

X= __________________ Y= __________________

8. ISA Design [12 pts]

Learn to use a new ISA that implements the following specification!

To cut hardware costs, a team of engineers has developed a simplified version of the LC2K ISA with only four registers—**buffer**, **RZR**, **RPR**, and **RSC**—which have the following functionality:

Register	**Description**
buffer	Used to move register values to/from memory via the instructions **loadbuf** and **storebuf**, or between different registers via the instructions **loadreg** and **storereg**
R**Z**R (“**Z**ero Reg”)	Holds the value 0 and cannot be changed
R**P**R (“**Pr**imary Reg”)	Holds values for compute, akin to standard LC2K registers
R**SC** (“**Sec**ondary Reg”)	Holds values for compute, akin to standard LC2K registers

This new ISA[1] has the following instructions and semantics:

Assembly Code	**Execution Semantics**	**Usage Example**
addi **imm.**	**RPR** = **RPR** + **imm.** (Add **imm.** value to **RPR**)	addi 10
add	**RPR** = **RPR** + **RSC**	add (**No args**)
nandi **imm.**	**RPR** = **RPR** NAND **imm.** (Nand **imm.** value with **RPR**)	nandi 42
nand	**RPR** = **RPR** NAND **RSC**	nand (**No args**)
beq **label**	Branch to **label** if **RPR** == **RSC**	beq end_label
loadreg **reg**	**reg** = **buffer** (**reg** can be **RPR** or **RSC**)	loadreg RPR
storereg **reg**	**buffer** = **reg** (**reg** can be **RPR, RSC,** or **RZR**)	storereg RZR
loadbuf **imm.**	**buffer** = mem[**imm.**] (load from **imm.** address)	loadbuf 32
storebuf **imm.**	mem[**imm.**] = buffer (store to **imm.** address)	storebuf 64

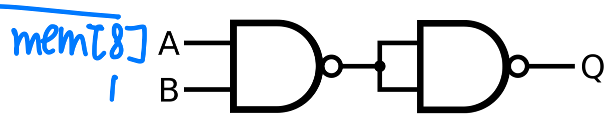
(a) **[2 pts]** If each instruction is encoded in 16 bits, and the addresses in **loadbuf** and **storebuf** instructions are unsigned immediates, what is the maximum number of address bits that can be in this ISA? (Assume opcodes are the same size in all instructions) What is the highest address a load or store instruction could refer to?

Maximum number of address bits: ______________

Highest loadable/storable address: ______________

[1] A copy of ISA can be found in the supplemental material for ease of reference.

(b) **[5 pts]** Use the new ISA to implement the C pseudocode on the left. Solutions exist that don't use all lines. The variable `mem` is an array of 32-bit values, thus each index corresponds to a single memory address. (*Hint*: NAND gates can create AND gates as shown in this diagram):



$[(A\cdot B)'(A\cdot B)']'$

$(A\cdot B)+(A\cdot B) => A\cdot B$.

C Pseudocode	New ISA

`if (mem[8] & 1) {`
`    Branch to label "goal"`
`}`
	`loadbuf 8` buf = mem[8]
	`loadreg RPR` RPR= mem[8]
	nandi #1 → RPR
	store buf RPR
	loadreg RSC
	nand → RPR.
	`storereg RZR` → buf = 0
	load reg RSC .
	`beq end`
	load reg RPR ← 若 RPR≠0.
	`beq goal` 让 RPR=0.
	`end`

(c) **[5 pts]** Use the new ISA to implement the C pseudocode on the left. Solutions exist that don't use all lines. The variable `mem` is an array of 32-bit values, thus each index corresponds to a single memory address.

C Pseudocode x4	New ISA

`mem[100] = (mem[64] + 3) << 2;`	`loadbuf 64` buf =mem[64]
	`loadreg RPR` RPR =mem[64]
	addi #3 RPR=mem[64]+3.
	storereg RPR buf=RPR.
	loadreg RCS RCS=RPR.
	add
	Storereg RPR
	load reg RCS
	add
	`storereg RPR`
	`storebuf 100`

***uniqname*: ______________**

9. Convert C to ARM Assembly [12 pts]

Implement this C code in ARM assembly by completing the missing instructions.

Convert this C code to ARM assembly by completing the ***twelve missing instruction parts***. Registers **X0** and **X1** hold the starting addresses of **set1** and **set2**, while registers **X2**, and **X3** hold the length of each set, respectively. The input argument of **search_value** is mapped to **X5** and return value of **find_mutual_sum** is at **X7**. Note that both set1 and set2 include **signed values**.

C

```
        64                                 X0
int64_t set1[] = [1, 5, -3, -8, ...];
int32_t set2[] = [-3, 2, -4, 5, ...]; X1
int64_t len1; X2 // length of set1
int64_t len2; X3 // length of set2

int64_t search_value(int64_t input);
                              X5
int64_t find_mutual_sum()
{                  X7                  X11 = X10×8
    int64_t sum = 0;
    int idx1 = 0; X10  X10    X2
    for ( ; idx1 < len1 ; idx1++) {
                  X5
        int64_t input = set1[idx1]; X11
        if (search_value(input) > 0)
            sum += input;
    }                  X7 = X7 + input.
    return sum;
}          X7


int64_t search_value(int64_t input) {
    int idx2 = len2 - 1;
       X12     X3
    for ( ; idx2 >= 0 ; idx2--) {
                        X6       4
        int64_t search =
            (int64_t) set2[idx2];
                           W
           X6
        if (search == input)
            return 1;
                    X8
    }
    return 0;
}
```

ARM

```
FIND_MUTUAL_SUM:
        MOVZ    X7,     #0,     LSL 0
        MOVZ    X10,    #0,     LSL 0
FIND_LOOP:
        CMP     X10,    X2
        B.GE    END_FIND_LOOP
        LSL     X11,    X10,    #3
        ADD     X11,    X11,    X0
        LDUR    X5,     [X11,   #0]
        BL      SEARCH_VALUE
        CBZ     X8,     CONT_FIND_LOOP
        ADD     X7,     X7,     X5,
CONT_FIND_LOOP:
        ADDI    X10,    X10,    #1
        B       FIND_LOOP

SEARCH_VALUE:
        MOVZ    X8,     #0,     LSL 0
        SUBI    X12,    X3,     #1
SEARCH_LOOP:
        CMP     X12,    XZR
        B.LT    Return
        LSL     X13,    X12,    #2
        ADD     X13,    X13,    X1
        LDURSW  X6,     [X13,   #0]
        CMP     X6,     X5
        B.EQ    IF_ELSE_SEARCH
        SUBI    X12,    X12,    #1
        B       SEARCH_LOOP
IF_ELSE_SEARCH
        ADDI    X8,     XZR,    #1
Return:
        BR      X30
END_FIND_LOOP:
```